Push-relabel based algorithms for the maximum transversal problem

Kamer Kaya, Johannes Langguth, Fredrik Manne, Bora Uçar

Push-relabel based algorithms for the maximum transversal problem

Kamer Kaya[a], Johannes Langguth[b], Fredrik Manne[c], Bora Uçar[d,e,*]

[a]*Department of Biomedical Informatics, The Ohio State University, Columbus OH, USA*
[b]*Simula Research Laboratory, Fornebu, Norway.*
[c]*Department of Informatics, University of Bergen, Norway*
[d]*CNRS*
[e]*LIP, ENS, Lyon, 69007, France. Tel: +33(0)472728932. Fax: + 33(0)472728080*

Abstract

We investigate the push-relabel algorithm for solving the problem of finding a maximum cardinality matching in a bipartite graph in the context of the maximum transversal problem. We describe in detail an optimized yet easy-to-implement version of the algorithm and fine-tune its parameters. We also introduce new performance-enhancing techniques. On a wide range of real-world instances, we compare the push-relabel algorithm with state-of-the-art algorithms based on augmenting paths and pseudoflows. We conclude that a carefully tuned push-relabel algorithm is competitive with all known augmenting path-based algorithms, and superior to the pseudoflow-based ones.

Keywords: Bipartite graphs, matching, push-relabel-based algorithms, graph theory

1. Introduction

The maximum cardinality bipartite matching problem is a classical topic in combinatorial optimization. Given a bipartite graph, the problem asks for a set of edges with maximum cardinality where no two edges in the set shares a vertex. The problem is polynomial time solvable. The algorithm with the best worst-case asymptotical complexity runs in $\Theta(\sqrt{\nu}\tau)$ time for a bipartite graph with ν vertices and τ edges [1]; for dense graphs, Alt et al. [2] present an algorithm with an improved complexity of $\Theta(\nu^{1.5}\sqrt{\tau/\log\nu})$. We study the currently best algorithms in order to determine the most efficient one in practical problems arising from real-world applications.

*Corresponding author
Email addresses: kamer@bmi.osu.edu (Kamer Kaya), jlangguth@simula.no (Johannes Langguth), Fredrik.Manne@ii.uib.no (Fredrik Manne), bora.ucar@ens-lyon.fr (Bora Uçar)

The maximum cardinality bipartite matching problem has numerous applications in diverse domains, including scheduling [3], timetabling [4, 5], image processing [6], and chemical structure analysis [7]. Burkard et al. [8, Ch. 3] treat the problem in great detail and portray two applications, one in vehicle scheduling and one in satellite telecommunications systems. The problem also arises in solvers for linear systems of equations. We are motivated by this last application, as algorithms for computing a maximum cardinality bipartite matching are run routinely in these solvers.

Consider the solution of a set of sparse linear equations on the form $\mathbf{A}x = b$, where $\mathbf{A}$ is a square, nonsingular matrix of coefficients. Given such a system, one first tests if $\mathbf{A}$ is reducible (see, e.g., KLU [9]). If this is the case, $\mathbf{A}$ can be permuted into the block triangular form (BTF), and the linear system can be solved with substantial computational savings [10, 11]. The BTF of $\mathbf{A}$ is obtained in two steps. In the first one, a maximum transversal—a set of nonzeros no two in the same row or column—is found in the matrix. This corresponds to finding a maximum matching in the bipartite graph corresponding to $\mathbf{A}$ (this bipartite graph contains a set of vertices for the rows and another set of vertices for the columns of $\mathbf{A}$ and an edge for each nonzero of $\mathbf{A}$). In the second step, the matrix is columnwise permuted to have the nonzero entries of the transversal to be on the main diagonal. Then, an algorithm to find the strongly connected components is run on the directed graph corresponding to the permuted matrix (in this graph, there is a vertex for each row/column, and the edge (i, j) for $i \neq j$ corresponds to an off-diagonal nonzero). The computationally heavy part of the whole process is the first step, as the second one can be implemented in linear time. Therefore, it is important to have the best performing algorithms for the bipartite matching problem.

There are several different algorithms for computing maximum matchings in bipartite graphs. One class of algorithms is based on augmenting paths. Duff et al. [12] discuss the design, analysis and implementation of eight augmenting path-based algorithms. Push-relabel-based algorithms [13] form a second class. A third class, pseudoflow algorithms, is based on more recent work [14] whose implementations are described by Chandran and Hochbaum [15].

Our contributions in this study are threefold. First, we present the push-relabel algorithm for the maximum cardinality matching problem in bipartite graphs in its elegance and simplicity. As the push-relabel algorithm was designed for the maximum flow problem, its usual presentations are much more complicated than necessary for the maximum bipartite matching problem. We give a pseudocode that is easy to implement and avoids unnecessary complexities. Our second contribution is a careful implementation and experimental comparison of the `FIFO` push-relabel algorithm with the currently best alternatives. Our experiments focus on maximum transversal problems arising in real-world applications. We report thorough results on all large enough problems corresponding to matrices from the University of Florida Sparse Matrix Collection [16]. Our third contribution is the adaptation of a simple strategy proposed by Duff et al. [12] to the push-relabel algorithm, as well as an additional modification, which speed up the algorithm noticeably.

In an accompanying technical report [17], we investigated the performance of the different push-relabel-based algorithms and concluded that the `FIFO` version is the best performing one. A preliminary version [18] of the current article contains a large set of experiments with the `FIFO` version in comparison with augmenting path- and pseudoflow-based algorithms. We will use the results presented in these reports to short-cut some experimental investigations (the reader will be alerted to check the reports whenever necessary).

The rest of this paper is organized as follows. We give the notation and the background in Section 2 and present the push-relabel (`PR`) algorithm in Section 3. Detailed descriptions of the different techniques used in the `PR` variants can be found in Section 4. In Section 5, we describe other algorithms used for comparison. Starting from Section 6, we present our experimental results, along with their discussion and conclusions in Section 7.

2. Notation and background

In a bipartite graph $G = (V_1 \cup V_2, E)$, the vertex sets V_1 and V_2 are disjoint and for all edges in E, one of the endpoints belongs to V_1 and the other belongs to V_2. For a vertex $v \in V_1 \cup V_2$, the *neighborhood* of v is defined as $\Gamma(v) = \{u : \{u, v\} \in E\}$.

A subset $\mathcal{M}$ of E is called a *matching* if a vertex in $V = V_1 \cup V_2$ is in at most one edge in $\mathcal{M}$. A matching $\mathcal{M}$ is called *maximal*, if no other matching $\mathcal{M}' \supset \mathcal{M}$ exists. A vertex $v \in V$ is *matched* (by $\mathcal{M}$) if it is in an edge in $\mathcal{M}$; otherwise, it is *unmatched*. A maximal matching $\mathcal{M}$ is called *maximum* if $|\mathcal{M}| \geq |\mathcal{M}'|$ for every matching $\mathcal{M}'$ where $|\mathcal{M}|$ is the cardinality of $\mathcal{M}$. Furthermore, if $|\mathcal{M}| = |V_1| = |V_2|$, $\mathcal{M}$ is called a *perfect* matching. The *deficiency* of a matching $\mathcal{M}$ is the difference between the cardinality of a maximum matching and $|\mathcal{M}|$. A good discussion on matching theory can be found in Lovasz and Plummer's book [19].

For a given $m \times n$ matrix $\mathbf{A}$, we define $G_{\mathbf{A}} = (V_R \cup V_C, E)$ where $|V_R| = m$, $|V_C| = n$, and $E = \{\{i, j\} \in V_R \times V_C : a_{i,j} \neq 0\}$ as the bipartite graph derived from $\mathbf{A}$. Assuming $\mathbf{A}$ is a square matrix having full structural rank, $G_{\mathbf{A}}$ has a perfect matching, and a *transversal* in $\mathbf{A}$ corresponds to a perfect matching $\mathcal{M}^*$ in $G_{\mathbf{A}}$. Based on this correspondence, we adopt the term *column* for a vertex in V_C and *row* for a vertex in V_R. The number of edges in $G_{\mathbf{A}}$ is equal to the number of nonzeros in $\mathbf{A}$ and denoted by τ.

We use two common data structures for storing sparse matrices [11, Section 2.7] to store the bipartite graphs in our implementations of the matching algorithms. These are called the compressed column storage (CCS) and the compressed row storage (CRS). They store edges of the bipartite graph as the neighborhoods of column or row vertices, respectively. Consider an $m \times n$ sparse matrix $\mathbf{A}$ with τ nonzeros. In CCS, the pattern of $\mathbf{A}$ is stored in two arrays:

- $rids[1, \ldots, \tau]$: stores the row index of each nonzero entry. The nonzeros in a column are stored consecutively.

- $cptrs[1, \ldots, n+1]$: stores the location of the first nonzero of each column in array $rids$ where $cptrs[n+1] = \tau + 1$. The row indices of the nonzeros in column j are stored in $rids[cptrs[j], \ldots, cptrs[j+1] - 1]$.

We refer to $rids$ and $cptrs$ as the CCS arrays. The CRS of a matrix $\mathbf{A}$ is the CCS of its transpose and vice versa. In CRS, there are again two arrays $cids$ and $rptrs$, of size respectively τ and $m + 1$, with functions similar to those of the above.

2.1. Maximum cardinality matching algorithms for bipartite graphs

Let $\mathcal{M}$ be a matching in G. A path in G is *$\mathcal{M}$-alternating* if its edges alternate between those in $\mathcal{M}$ and those not in $\mathcal{M}$. An $\mathcal{M}$-alternating path $\mathcal{P}$ is called *$\mathcal{M}$-augmenting* if the start and end vertices of $\mathcal{P}$ are both unmatched. The following theorem is a basis for the augmenting path-based algorithms for the maximum matching problem in the literature.

Theorem 1 ([20]). *Let G be a graph (bipartite or not) and let $\mathcal{M}$ be a matching in G. Then $\mathcal{M}$ is of maximum cardinality if and only if there is no $\mathcal{M}$-augmenting path in G.*

There are three prominent classes of bipartite matching algorithms: augmenting path-based ones, push-relabel-based ones, and the recently proposed pseudoflow-based ones. Below we briefly mention the main characteristics of these algorithms, and defer further details to later sections.

Algorithms based on augmenting paths follow a common pattern. Given a possibly empty matching $\mathcal{M}$, this class of algorithms searches for an $\mathcal{M}$-augmenting path $\mathcal{P}$. If none exists then the matching $\mathcal{M}$ is maximum by Theorem 1. Otherwise, the alternating path $\mathcal{P}$ is used to increase the cardinality of $\mathcal{M}$ by setting $\mathcal{M} = \mathcal{M} \oplus E(\mathcal{P})$ where $E(\mathcal{P})$ is the edge set of a path $\mathcal{P}$, and $M \oplus E(\mathcal{P}) = (\mathcal{M} \cup E(\mathcal{P})) \setminus (\mathcal{M} \cap E(\mathcal{P}))$ is the symmetric difference. This inverts the membership in $\mathcal{M}$ for all edges of $\mathcal{P}$. Since both the first and the last edge of $\mathcal{P}$ were unmatched in $\mathcal{M}$, we have $|\mathcal{M} \oplus E(\mathcal{P})| = |\mathcal{M}| + 1$. The way in which augmenting paths are found constitutes the main difference between the algorithms based on augmenting path search, both in theory and in practice. In this paper, we use `PFP` (described in Section 5.1), the fastest augmenting path-based matching algorithm identified by Duff et al. [12].

Push-relabel algorithms on the other hand search and augment simultaneously. They do not explicitly construct augmenting paths. Instead, they repeatedly augment the prefix of a speculative augmenting path $\mathcal{P}_2 = (v, u, w)$ in G where u is matched to w, and $v \in V_C$ is an unmatched column. Augmentations are performed by unmatching w and matching v to u. If the neighbor of an unmatched column is also unmatched, the suffix of an augmenting path has been found, allowing the augmentation of $|\mathcal{M}|$. The speculative augmentation operations are performed until no further suffixes can be found. These operations are guided by assigning a label to every vertex which provides an estimate of the distance to the nearest unmatched row (i.e., to a potential suffix).

The original push-relabel algorithm by Goldberg and Tarjan [13] was designed for the maximum flow problem. Since bipartite matching is a special case of maximum flow, it can be solved by this algorithm. In fact, it is known to be one of the fastest algorithms for bipartite matching [21]. In this paper, we study the performance of the best variant identified in our technical report [17]. We will discuss the simple push-relabel algorithm (`PR`) and its extensions in detail in Sections 3 and 4.

The pseudoflow-based [14] bipartite matching algorithms progress in a way similar to `PR`. In the matching context [15], they can be described as building trees containing prefixes and suffixes of augmenting paths (see Section 5.2). When a prefix- and a suffix-tree connect, an augmenting path is found. Different variants of the pseudoflow algorithm differ in the size of the trees constructed, as well as in the fashion of constructing them.

For a short summary on some other algorithms and approaches for the bipartite graph matching problem, we refer the reader to a recent survey [12, Section 3.4].

2.2. Initialization heuristics

Almost all matching algorithms start with an empty matching and find matchings of successively increasing size in some fashion, the algorithms studied here being no exception. These successive steps are self-contained. Thus, these algorithms can be initiated with a non-empty matching. In order to exploit this, several efficient and effective heuristics, which find initial matchings of considerable size, have been proposed in the literature [22, 23, 24, 25, 26].

In this paper, we use two different initialization heuristics. The first one, which we call simple greedy matching (`SGM`), examines each unmatched column $v \in V_C$ in turn and matches it with an unmatched row $u \in \Gamma(v)$, if such a row exists. Although it is the simplest available heuristic, `SGM` is probably the most frequently used one in practice. The second heuristic is `KSM`, proposed by Karp and Sipser [23]. It is similar to `SGM`, but it keeps track of the vertices with a single unmatched adjacent vertex and immediately matches these. Theoretical studies by Aronson et al. [27] and Karp and Sipser [23] show that `KSM` is highly likely to find perfect matchings in random graphs, and in practice, it is significantly more effective than `SGM`. The `SGM` algorithm needs only CCS (or CRS), however, `KSM` needs both data structures.

These heuristics have seen extensive experimental investigations, among others by Duff et al. [12], Langguth et al. [24], and Magun [25]. There are extended versions of these heuristics [24, 25]. However, none of the extended heuristics could be shown to consistently provide performance superior to `KSM` or `SGM`. Therefore, only these last two heuristics are considered here.

3. The push-relabel algorithm for bipartite matching

Cherkassky et al. [21] describe the (simplified) push-relabel algorithm for the bipartite matching problem. In the following, we carefully portray this

algorithm in a ready-to-implement pseudocode form as shown in Algorithm 1. This algorithm will be referred to as PR throughout the paper. Section 4 contains several extensions of PR which were used in the experiments.

Algorithm 1 PR: Push-Relabel Algorithm for Bipartite Matching

Input: A bipartite graph $G = (V_R \cup V_C, E)$ and a (possibly empty) matching $\mathcal{M}$
Output: A maximum cardinality matching $\mathcal{M}^\star$

```
1: Set ψ(u) = 0 for all u ∈ V_R
2: Set ψ(v) = 1 for all v ∈ V_C
3: Set all v ∈ V_C unmatched by M to active
4: while an active column v exists do
5:    Find a row u ∈ Γ(v) of minimum ψ(u)
6:    if ψ(u) < m + n then
7:       ψ(v) ← ψ(u) + 1          ▶ Relabels v if {u, v} is not an admissible edge
8:       if {u, w} ∈ M then
9:          M ← M \ {u, w}        ▶ Double push
10:         Set w active
11:      M ← M ∪ {u, v}           ▶ Push
12:      ψ(u) ← ψ(u) + 2          ▶ Relabels u to obtain an admissible incident edge
13:   Set v inactive
14: return M* = M
```

Let $\psi : V_R \cup V_C \to \mathbb{N}$ be a distance labeling used to estimate the distance and thereby the direction of the closest unmatched row for each vertex. This labeling constitutes a lower bound on the length of an alternating path from a vertex v to an unmatched row. If v is an unmatched column, such a path is also an augmenting path. During initialization, the algorithm sets $\psi(v) = 1$ for all $v \in V_C$ and $\psi(v) = 0$ for all $v \in V_R$. We call unmatched columns active. Now, as long as there are active columns, the algorithm repeatedly selects one of them and performs the *push* operation on it.

To perform a push on an unmatched column v, we search $\Gamma(v)$ for a row $u \in \Gamma(v)$ with the minimum $\psi(u)$. Note that $\psi(v) - 1$ is the infimum for the value of $\psi(u)$. This holds after the initialization ($\psi(u) = 0$ and $\psi(v) = 1$ for all $u \in V_R$ and $v \in V_C$) and is maintained throughout the algorithm as an invariant. As soon as an edge $\{v, u\}$ having $\psi(v) = \psi(u)+1$ is found, the search stops. Such an edge is called *admissible*.

If $u \in \Gamma(v)$ has minimum ψ and is not matched, it can be matched to v immediately by adding $\{v, u\}$ to $\mathcal{M}$ and thereby increasing the cardinality of $\mathcal{M}$ by one. This operation is called a *single push*. On the other hand, if u is matched to a column vertex w, we perform a *double push*. This operation removes $\{w, u\}$ from $\mathcal{M}$, adds $\{v, u\}$ to $\mathcal{M}$, and makes w active. The double pushes ensure that once a row is matched, it can never become unmatched again—the cardinality of $\mathcal{M}$ can never decrease. An unmatched row vertex u has $\psi(u) = 0$, and it will always have the minimum ψ value.

If there is no admissible row u among the neighbors of v, i.e., any row u having minimum $\psi(u)$ has $\psi(u) > \psi(v) - 1$, we set $\psi(v)$ to $\psi(u) + 1$. This is referred to as a *relabel* on v. Clearly, doing so does not violate the above

invariant due to the minimality of $\psi(u)$. To understand the motivation for a relabel on v, remember that $\psi(u)$ is a lower bound on the length of an alternating path from u to a closest unmatched row. Now, even though no path between v and its closest unmatched row necessarily contains u, it must contain some $u' \in \Gamma(v)$. Since $\psi(u)$ was the minimum among the labels of all the neighbors of v, we have $\psi(u') \geq \psi(u)$. Thus, $\psi(u) + 1$ is a lower bound on the length of a path between v and its closest unmatched row, and $\psi(v)$ is updated accordingly.

By the same token, u is relabeled by increasing $\psi(u)$ by two following a push. For a single push, this means that we have $\psi(u) = 2$ now. Since G is bipartite and u is no longer an unmatched row, it is clear that the distance to the next unmatched row must be at least two after a single push. In case of a double push, any alternating path from u to a closest unmatched row now contains v. As any such path starts with an unmatched edge on an unmatched row and G is bipartite, the path contains only matched edges going from columns to rows and only unmatched edges from rows to columns. Thus, the actual distance for u must be at least $\psi(v)+1$. Because $\psi(v)$ was either relabeled to $\psi(u)+1$ prior to the push or had this value to begin with, increasing $\psi(u)$ by two yields a correct new lower bound. Clearly, this increase maintains the invariant $\psi(u) \geq \psi(v)-1$.

When implementing the push-relabel algorithm, we can eschew storing the row labels, since $\psi(u)$ will always be either 0 if u is unmatched, or equal to $\psi(w) + 1$ if u is matched to w.

If $\psi(u) \geq m + n$ for the minimum $\psi(u)$ among the neighbors of v, instead of performing a push or relabel, v is considered unmatchable and marked as inactive. The reason for this is that the maximum length of any augmenting path in G is at most $\min(2m, 2n) - 1$. Since ψ is a lower bound on the length of a path to an unmatched row, and $\psi(u) \geq m + n$ for all neighbors of v, no augmenting path can start at v. As v remains unmatched, it can never become active again via a double push. Thus, it will not be considered any further by the algorithm.

The push and relabel operations are repeated until there is no active vertex left, either because all vertices have been matched or marked as inactive. Using Theorem 1, it is easy to show that in this case $\mathcal{M}$ is a maximum matching. The time complexity of the algorithm is $\mathcal{O}(n\tau)$ [13].

As discussed above, one needs to store the column labels. In order to reduce jumps and arithmetic operations, we store the row labels as well. Our implementation therefore uses $m + n$ integer space in addition to the CCS arrays. We also keep the matching partners of rows and columns in arrays, requiring an additional $m + n$ space.

4. Modifications to the push-relabel algorithm

We now consider several modifications to the basic push-relabel algorithm in order to optimize its performance. The modifications include applying a strict order of push operations and heuristics that update the distance labeling ψ. Both are well studied in the literature [21, 28]. We also present new techniques inspired by the augmenting path algorithms.

4.1. Push order

The push-relabel algorithm repeatedly selects an active column on which it performs a push operation. The order in which active columns are selected is not fixed; any implementation needs to define an order. A simple solution for this is to maintain a stack or queue of active columns and select the first or topmost element, resulting in `LIFO` (last-in-first-out) or `FIFO` (first-in-first-out) push order. Alternatively, each active column v can be sorted into a priority queue according to its label $\psi(v)$. We restrict ourselves to `FIFO` ordering which was found to be superior (see the technical report [17]). An additional memory space of size n is required to implement the `FIFO` ordering, making the total memory requirement $m+2n$ integers (on top of the CCS and matching arrays).

4.2. Global relabeling

The performance of the `PR` algorithm can be improved by periodically setting all labels to exact distances. This is called *global relabeling* and is accomplished by running a BFS starting from the unmatched rows, as shown in Algorithm 2. The label of each vertex v visited by the BFS is set to the minimum distance from v to any unmatched row. Each vertex w not visited by the BFS is assigned a label $\psi(w) = m + n$, thereby removing it from further consideration.

In order to keep track of the number of pushes executed, a counter is incremented every time the value of $\psi(v)$ is changed in Line 7 of Algorithm 1. Thus, pushes along admissible edges are not counted. Note that the single pushes are always along admissible edges. When the counter reaches a predetermined threshold, we call the `Global Relabeling` procedure. A threshold of n was suggested as the standard frequency of global relabels [21].

Algorithm 2 : Global Relabeling

Input: A bipartite graph $G = (V_C \cup V_R, E)$ and a matching $\mathcal{M}$ in G
Output: An accurate distance labeling ψ w.r.t. $\mathcal{M}$

```
1:  Q ← u for all unmatched u ∈ V_R
2:  Set ψ(v) = m + n for all v ∈ V_C
3:  Set ψ(u) = m + n for all matched u ∈ V_R
4:  while Q not empty do
5:      u ← POP u from Q
6:      for all v ∈ Γ(u) do
7:          if ψ(v) = m + n then
8:              ψ(v) ← ψ(u) + 1
9:              if {v, w} ∈ M then
10:                 ψ(w) ← ψ(v) + 1
11:                 PUSH w to Q
12: return ψ
```

Since our implementation makes use of the double push technique, we need to adopt a counting scheme that differs slightly from the standard `PR` algorithm. We only count the number of double pushes in which the first edge was not admissible. The second edge, which started out as matched, would always require

a relabel prior to a push, unless its label was changed by a global relabeling between it becoming matched and the current double push. Since the row vertex is relabeled immediately after a double push, it is impossible to accurately reflect this in the count without performance degradation. Therefore, we count double pushes only once and reflect this in the thresholds. The single pushes always use admissible edges, and are therefore never counted against the threshold.

Since we use rectangular matrices as test instances, we must consider the case $m \neq n$. Preliminary experiments showed no noticeable difference between using relabeling frequencies of m and n. Thus, we use a base threshold of $m+n$. The corresponding relabeling frequency is denoted as `RF=1`. Multiples of this base relabeling frequency `RF=1.5`, `RF=2`, `RF=4`, and `RF=8` are also suggested [17, 21].

The global relabeling operation requires another array of size m to maintain a queue of the discovered row vertices. As the BFS is run from the row vertices, the CSR storage is also required. Therefore, in addition to CCS, CSR, and two matching arrays, a total of $2m + 2n$ integer space is required to implement `PR-FIFO` with global relabeling.

4.3. Fairness

By default, our `PR` implementations always search through adjacency lists in the same order when seeking a neighbor of minimum ψ. This raises the question of whether the algorithm could be improved by encouraging fairness in neighbor selection. This was proposed by Duff et al. [12] for improving the Pothen and Fan (`PF`) algorithm [22] and resulted in significant performance gain (discussed in Section 5.1). By varying the direction of search through the adjacency list of an active column, the likelihood of the algorithm repeatedly pursuing an unpromising direction of search is reduced.

The fairness technique can also be applied in the `PR` algorithm. We study this `Fair` variant and compare it to other `PR` implementations in Section 6. No extra storage is required for this technique.

4.4. Search spread

In push-relabel algorithms designed for the maximum flow problems, a different technique is used to equilibrate searches over the adjacency lists. In our setting, this can be described as follows. Every vertex v maintains a pointer $p(v)$ which is set to its first incident edge on initialization. The search for a neighbor of minimum label always starts with the edge to which $p(v)$ points. If an admissible edge is found, the search is stopped and $p(v)$ is set to the next edge in the list of edges incident to v. This guarantees that the search is spread out more evenly among incident edges, making it more likely that an admissible edge is found quickly. If a search starting from $p(v)$ reaches the end of the adjacency list belonging to v without finding an admissible edge, it continues at the start of the adjacency list and proceeds up to $p(v)$. However, if a neighbor u with $\psi(u) = \psi(v) + 1$ is found, this latter part can be skipped since no admissible edge, and thus no neighbor with $\psi(u) < \psi(v) + 1$ exists. To see this, remember that a neighbor u having $\psi(u) = \psi(v) - 1$ implies an admissible edge $\{v, u\}$, and that $\psi(u)$ is always incremented by 2.

We combined fairness with the search spread technique, obtaining the `Fair-Spread` variant of push relabel. This leads to a somewhat more complicated implementation because $p(v)$ now switches between acting as the starting point and the endpoint of a search. In order to improve clarity of the code, we implemented the combined technique described above using two additional arrays of size n each. However, only one additional array is required.

5. Other algorithms

Here, the algorithms that we experimentally compare with `PR-FIFO` are described. We use only the fastest known algorithms for our experiments: `PFP` and pseudoflow-based algorithms. `PFP` is the best performing augmenting path-based algorithm [12, 17]. Chandran and Hochbaum [15] found the *free arcs* variant of the pseudoflow algorithms to be superior for the bipartite matching problem on difficult instances. We investigate this variant and two other variants which were found to be very efficient [15]. Our descriptions are necessarily brief; the reader is referred to the original resources [1, 2, 12, 15, 22, 29].

5.1. `PFP`*: A matching algorithms based on augmenting paths*

The Pothen-Fan algorithm, denoted as `PF`, is based on repeated phases of depth-first searches [22]. At a phase, `PF` performs a maximal set of vertex disjoint DFSs, each starting from a different unmatched column. A vertex can only be visited by one DFS during each phase. Any DFS that succeeds in finding an unmatched row immediately suggests an augmenting path. As soon as all the searches have terminated, the current matching is augmented along all the augmenting paths found in this manner. After this, a new phase starts. The algorithm has a running time of $\mathcal{O}(n\tau)$.

In each DFS, the rows adjacent to a column are visited according to their order in the adjacency list. This is true even if there is an unmatched row among them. In order to reach such an unmatched row, a pure DFS-based algorithm may need to explore a large part of the graph and hence may be very costly. To alleviate this problem, a mechanism called *lookahead* is used [22, 30]. This mechanism enables keeping track of unmatched rows in an adjacency list.

Duff et al. [12] found `PF` to be efficient for matrices from real-world applications, except that its running time varies widely for different ordering of rows or columns. To alleviate this, they suggested using the *fairness* technique (see Section 4.3). This technique does not change the complexity or the memory requirements of `PF`. It usually improves the performance of `PF`, and in some cases it results in remarkable speedups, while the required overhead remains negligible [12]. We use this variant of `PF` exclusively and refer to it as `PFP`.

The implementation of `PFP` given by Duff et al. [12] uses integer arrays of total size $m + 4n$ in addition to the CCS and matching arrays. The algorithm does not need CRS itself. However, we always initialize it using `KSM`, which needs both CCS and CRS.

5.2. The pseudoflow algorithm

The pseudoflow algorithm was introduced by Hochbaum [14] for the maximum flow problem. It incorporates notions developed for the push-relabel algorithm, among them the distance labeling ψ and the admissible edge definition. In the bipartite matching context, the algorithm can be simplified. We describe the simplified free-arcs variant of the pseudoflow algorithm, as this one was reported previously [15] to be the fastest variant for the bipartite matching problem.

All vertices v start out as unmatched and having $\psi(v) = 1$. Similar to `PR`, unmatched columns are marked as active and processed in a given order, e.g., the lowest label first. An active vertex v scans its adjacency list for admissible edges and, if necessary, increases its label $\psi(v)$ such that an edge leading to a lowest labeled neighbor becomes admissible. Let $\{v, u\}$ be an admissible edge found in this manner. If u is unmatched, v and u are matched along $\{v, u\}$ and v becomes inactive. This is equivalent to a single push. Otherwise, u becomes overmatched. Let w be the original matching partner of u. Unlike during a double push in the `PR` algorithm, w now remains matched to u. Next, v and w both become active and $\psi(u)$ is increased to $\psi(v) + 1$.

Now assume w (or equivalently v) is processed and an admissible edge $\{w, x\}$ is found. If x is unmatched, then $\{w, u\}$ becomes unmatched, while $\{w, x\}$ becomes matched, resulting in two standard matching edges $\{v, u\}$ and $\{w, x\}$. If x was already matched to some other vertex y, $\{v, u\}$ becomes a standard matching edge while v, x, and y now form a new path of length two where v and x are active.

The process continues until all active vertices have been matched along standard matching edges, thereby becoming inactive or their labels have increased to $m+n$. Similar to `PR`, an active vertex v with $\psi(v) = m+n$ is set to inactive. If no active vertices remain, the algorithm terminates. The running time of this algorithm is $\mathcal{O}(n\tau)$.

In addition to the free-arcs variant, several alternatives are described by Chandran and Hochbaum [15]. These variants are able to build larger trees than the length two paths described above. Similar to `PR`, the pseudoflow algorithm repeatedly processes active vertices. Thus, different strategies of selecting active vertices are possible. Both the highest-label-first and the lowest-label-first are used by Chandran and Hochbaum [15]. Active vertices are kept in buckets. The buckets can be implemented either as FIFO queues or as LIFO stacks. We select the highest-label-first variant with LIFO buckets, which is referred to as the `HI_WAVE` variant in [15]. We also use `LO_LIFO`, the lowest-label-first variant with LIFO buckets. The free-arcs variant is referred to as `LO_FREE`. Other variants were found to be inferior to these [31]. We confirmed this in preliminary experiments by studying the `HI_FIFO`, `LO_FIFO`, and `HI_FREE` variants. Overall performance was about 20% inferior to the `HI_WAVE`, `LO_LIFO`, and `LO_FREE` versions that we study in this paper.

At the time of writing, implementations of the pseudoflow algorithms were publicly available at `http://riot.ieor.berkeley.edu`. The implementation

given there uses eight fields (mixture of integers and pointers) per vertex to store one class of vertices (say row vertices) and uses ten fields (mixture of integer and pointers) per vertex to store the other class of vertices. Furthermore, adjacency lists are stored for the rows and for the columns. Four additional arrays, of size n (or m) each, are used during the algorithm. Overall, the total memory requirement is $8m + 14n + 2\tau$. We identified one field in each vertex class as redundant for our applications in sparse matrices; however we did not see an easy way to reclaim the space used by other fields and the four arrays. Therefore, we deem it accurate to state that the space requirement of a reasonable pseudoflow-based matching algorithm is $7m + 13n + 2\tau$.

6. Experiments

6.1. Experimental setup

All of the algorithms and heuristics are implemented in the C programming language and are available at `http://bmi.osu.edu/~kamer/research.html`. Codes are called via a Matlab interface. We compiled the codes with *mex* of Matlab using *gcc* version 4.4.2 with the optimization flag -O and ran the compiled codes on a machine with a 2.4 Ghz AMD Opteron 250 processor and 8 Gbytes of RAM. As an additional test system, an Intel Xeon E5520 Quad Core computer running at 2.27 Ghz was used. Differences in the results were marginal, and thus they are not presented here.

For the experiments, we use real-world $m \times n$ matrices from the University of Florida Sparse Matrix Collection [16]. We only consider matrices having more than 50000 columns. Due to the comparatively steep memory requirements of the pseudoflow codes, it was necessary to limit the maximum number of columns to 12 million and the maximum number of nonzeros to 120 million. Currently a total of 437 matrices satisfy these assertions, with 53 among them being rectangular. On average, the matrices have approximately 600000 rows and an equal number of columns and close to 10 million edges. The respective median values are 155000 and 2.7 million.

For each matrix, we perform four sets of experiments. First, we execute all algorithms on the original matrix (denoted by "No perm"). Next, we apply either a row, column, or both a row and a column permutation to each matrix before executing all of the algorithms. The respective results are labeled as "Row perm", "Col perm", and "Row+Col perm". For each algorithm, the average running time and operation counts of five permutations is stored as the running time or operation count of the algorithm on a matrix with a given permutation type.

Although our focus is on the maximum transversal problem for real-world instances, we perform smaller confirmation experiments (described later) on bipartite random instances.

6.2. Measurements

To compare the algorithms, we measure both running times and machine independent operation counts.

For PFP and PR, the total running time is divided into three parts. First $\mathbf{A}^T$, the transpose of the input matrix $\mathbf{A}$ must be computed in order to obtain the CRS array of $\mathbf{A}$. This is necessary for global relabelings in PR and for the KSM initialization heuristic. Thus, the first step requires an identical amount of work for PFP and PR algorithms. We then obtain the time required for the KSM or SGM initialization heuristic and finally the time for the main algorithm. The pseudoflow algorithms also perform three steps. The first step consists of building up sophisticated data structures, and thus requires considerably more effort than taking the transpose. The second step is always an SGM style initialization that works on these data structures. In the third step, the actual algorithm is called and its running time is measured.

Due to the different structure of the algorithms, not all operations are comparable across the algorithms. All algorithms repeatedly search through adjacency lists in order to check adjacent vertices. Therefore, such edge operations, which are commonly referred to in the literature as *Arc Fetches* or *Arc Scans* are counted for all algorithms. For PFP, the second relevant operation is obtaining the symmetric difference between the current matching $\mathcal{M}$ and an augmenting path. We refer to matching an edge while unmatching another as an augmentation. Since the number of augmenting paths to be used is equal to the deficiency of the initialization, the number of augmentations depends on the length of the augmenting paths found. We report this number for PFP.

For PR, instead of augmentations we have double pushes, which require slightly larger effort, as the labels have to be updated. Arc scans tend to be less expensive, as often the entire adjacency list of a vertex is scanned, which is cache-efficient. The same is true for arc scans performed during a global relabel, which progresses in a BFS fashion, as opposed to the DFS in PFP.

For the pseudoflow algorithms, we again report the number of arc scans. Also, the available codes report the number of additional operations. These operations are not directly comparable to double pushes, but each of them is at least as expensive as an arc scan.

Because the initialization heuristics usually match at least 95% of the vertices, the operation of matching a vertex for the first time, i.e., a single push in PR or its equivalent in the other algorithms, is not counted since its number is hardly significant.

6.3. Running time

The running time results over the test set are given in Table 1. We measure running time in seconds. Reported timings include the time to build up data structures, and heuristic initialization, but not file reading. We give both the average and the median running time. Since all algorithms have superlinear running times, the averages are significantly higher than the medians. The ratio of the average running time to the median running time is 10.54 on average (for details see [18, Table 5]). Algorithms for which the discrepancy between average and median is low can be regarded as stable with respect to different instances. In general, PR showed higher ratios than the other algorithms. Note however that the correlation between instance size and running time is rather weak.

Details on this correlation and on the average-median ratios can be found in the technical report version of this article [18, Fig. 4].

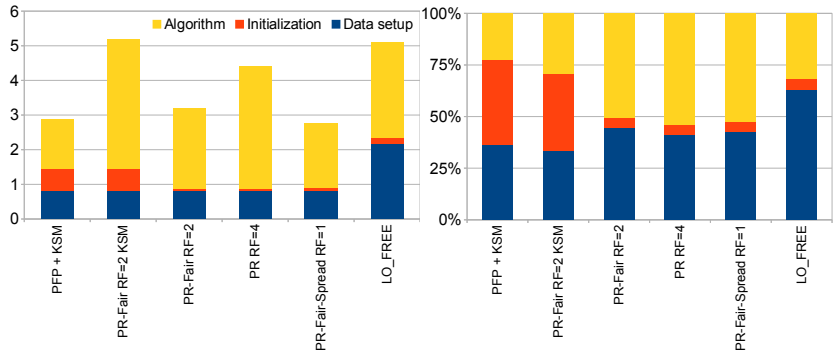
Table 1: Average and median running times in seconds over the entire test set, for various permutations. Values contain data setup, initialization and main algorithm time. Best average values (for a permutation type) are shown in boldface. Many median values are tied.

Algorithm	RF	No perm		Row perm		Col perm		Row + Col perm		Average	
		Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.
PR	1	1.13	0.10	3.41	0.32	10.15	0.51	**3.74**	0.44	4.61	0.31
SGM	1.5	1.05	0.10	3.47	0.30	9.79	0.49	3.86	0.44	4.54	0.30
	2	0.99	0.09	3.23	0.32	9.47	0.49	3.97	0.44	4.42	0.29
	4	0.94	0.09	3.24	0.40	4.27	0.52	4.32	0.57	3.19	0.34
	8	0.95	0.09	3.10	0.37	4.57	0.50	4.12	0.49	3.18	0.32
PR-Fair	1	1.11	0.10	4.90	0.32	4.36	0.37	4.93	0.45	3.82	0.28
SGM	1.5	1.00	0.09	4.94	0.30	4.35	0.37	5.04	0.45	3.83	0.27
	2	0.98	0.09	3.68	0.33	**3.12**	0.36	5.06	0.42	3.21	0.28
	4	**0.92**	0.09	3.16	0.37	3.88	0.40	4.19	0.53	3.04	0.30
	8	0.93	0.09	3.31	0.41	3.41	0.46	4.32	0.55	2.99	0.33
PRFair	1	1.12	0.10	2.88	0.32	3.25	0.39	3.77	0.45	**2.76**	0.28
-Spread	1.5	1.03	0.10	2.95	0.32	3.33	0.38	3.88	0.45	2.79	0.28
SGM	2	1.01	0.10	3.10	0.34	3.32	0.39	4.06	0.46	2.87	0.29
	4	0.96	0.09	3.10	0.37	3.94	0.49	4.03	0.49	3.01	0.32
	8	0.94	0.10	3.34	0.42	3.47	0.46	4.35	0.56	3.02	0.34
Other algs.											
PFP		1.21	0.13	**2.74**	0.36	3.79	0.48	3.86	0.57	2.90	0.36
HI_WAVE		5.95	0.81	6.83	1.05	7.72	1.16	6.17	0.90	6.67	0.97
LO_LIFO		6.71	0.76	9.47	1.14	9.38	1.24	6.99	0.85	8.14	1.00
LO_FREE		4.75	0.77	3.46	0.65	8.16	1.20	4.92	0.95	5.32	0.89

Overall, PR-Fair-Spread with low relabeling frequency shows the best results. It dominates PFP, and shows far better average and only slightly worse median results than PR-Fair. It is also superior to PR without fairness. However, all these algorithms are relatively close in performance. The pseudoflow codes show far lower performance. They also show smaller relative variance in running time. Due to the fact that KSM initialization consistently provides much better initializations than SGM, PFP shows a good average running time, but its median performance is lower than that of most PR codes.

Using KSM initialization is not competitive for PR. It leads not only to higher median running times, but also to very high average values. We investigated this in detail [17, p. 26], [18, Table 3]. We also observed that using SGM followed by a global relabeling is generally preferable to starting PR with an empty matching.

The PR algorithm is quite sensitive to the frequency of the global relabelings. If the fairness technique is not used, a setting of $RF = 4$ or $RF = 8$ is preferable. With fairness, we observe maximum performance at $RF = 2$ or lower, where the median performance is clearly improved. However, the best results are obtained by using search spread and $RF = 1$. Therefore, in the following we will focus on PR-Fair-Spread at $RF = 1$ and PR-Fair at $RF = 2$ when discussing the PR algorithm. We will also study PR without fairness at $RF = 4$. Interestingly, for the unpermuted matrices, setting $RF = 8$ yields the best results, while the



(a) Average time in seconds

(b) Median percentages

Figure 1: Comparison of the division of (a) average running time (in seconds) and (b) median percentages taken by different parts of the algorithms. In each bar, the lowest, middle, and highest segments correspond, respectively, to the time spent in data set up, initialization heuristic, and the main algorithm.

fairness and the search spread mechanisms have no noticeable effect.

Concerning permuted matrices, we observe that `PFP` is faster than `PR` on row permuted matrices, roughly equal on row-column permuted matrices, and slower on column permuted or original matrices. These differences are the result of the different algorithmic techniques. On original matrices, the BFS based global relabeling used in `PR` is obviously very effective. Increasing its frequency makes `PR` even faster. However, as it works on the row adjacency lists, `PR` is slower than `PFP` under pure row permutations, since `PFP` does not work with the row adjacency lists at all, except during `KSM` initialization.

On the other hand, the fact that `PFP` works only on the column adjacency lists makes it more susceptible to pure column permutations than `PR`, where augmentations are guided by labels which are updated during global relabelings. Still, `PR` is affected considerably by column permutations. Consequently, having both row and column permutations is the hardest case for `PR`, and its performance is lowest here. `PFP` is not affected by the addition of row permutations, and shows roughly the same performance as for pure column permutation, which is comparable to that of `PR` on such instances. Note that without fairness, `PR` shows very low performance under pure column permutations. In the absence of column permutations, fairness has little effect.

The division of running time for the different algorithms is illustrated in Figure 1. We observe that transposing the matrix consumes a significant amount of running time. However, as both `PR` and `PFP` require this step, it does not affect their relative comparison. Setting up the data structures for `LO_FREE` is even more expensive as shown in Figure 1a.

Next, we see that `KSM` initialization is quite expensive, taking up almost half

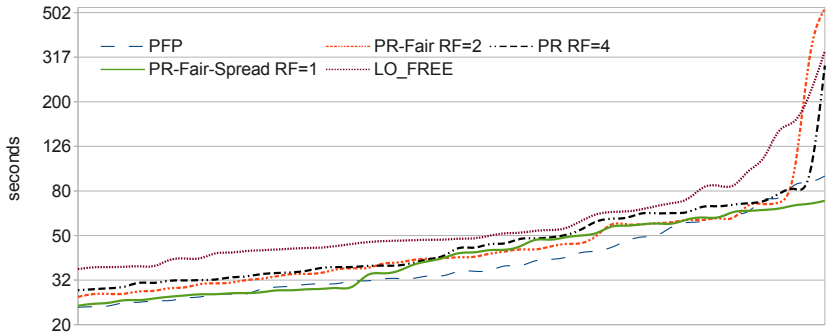


Figure 2: Comparison of the 50 most time consuming instances and permutations for each algorithm, sorted along the x-axis. Running time is given in seconds; grid lines are on a logarithmic scale. `PFP` and `PR-Fair-Spread` are very fast, even for their respective worst cases.

the time for the main computation of `PFP`. On the other hand, using `SGM` is very fast. However, `KSM` initialization results in `PFP` having the shortest main computation, making it competitive with the `Fair PR` codes while `LO_FREE` has the longest main computation time, rendering it uncompetitive.

In addition to the averages, we study the behavior on the worst cases, i.e., on the original matrices or permutations of them that are the most time consuming for each algorithm. Figure 2 indicates that on the 50 most difficult instances `PFP` is slightly faster than `PR`. `LO_FREE` starts about 50% slower, and its running time increases considerably on the hardest 5 instances. `PR` without the fairness mechanism is generally slower than `PR-Fair`, becoming the slowest on the most difficult instances. Both variants have very high running times on permutations of the instance *circuit5M* [18, Table 6]. Otherwise, the overall worst case picture closely matches the findings obtained from studying the average running times.

In order to study the instances that are solved quickly by most algorithms we give the performance profile of the algorithms in Figure 3. All `PR` algorithms have minimum running time over all algorithms in about 30% of the instances. For `PFP`, this figure is about 20%. `PFP` remains consistently slower than the `PR` algorithms by a small margin. Among these, we see that `PR-Fair` without `Spread` has a slightly better performance. This is not surprising since the `Spread` technique costs some additional overhead which only pays off in the worst cases, as seen in Figure 2. Consistent with its average performance, `LO_FREE` remains far slower than all alternatives. `LO_FREE`'s running time is smaller than twice the running time of the fastest algorithm in only about 20% of the cases, whereas the corresponding percentages for `PFP` and `PR-Fair` are 85% and 95%, respectively. In conclusion, we see that algorithms which show good performance in this profile also have low median running times in Table 1. Some interesting instances in which the slowest algorithm's running time is 100 times larger than those of the fastest one are reported in the technical report [18, Table 6].

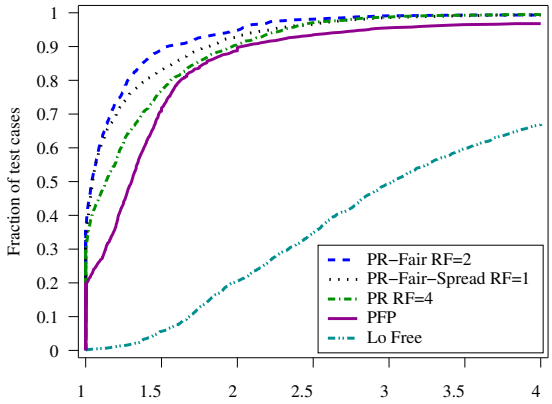


Figure 3: Performance profile for the principal algorithms. It denotes the fraction of instances for which an algorithm is within a given factor of the best algorithm for that instance. The factors are denoted on the x-axis, while the y-axis shows the fraction of instances.

6.4. Operation counts

The differences in operation counts for the various algorithms largely resemble the differences in running time. Average results over the entire test set are given in Table 2 (additional results with `PR` using `KSM` are in the technical report [18, Table 4]). We first observe that `PFP` requires a comparatively small amount of arc scans and matching operations. However, since this number does not include approximately τ arc scans required by the `KSM` initialization, we have to add the average number of edges in the test set, which is approximately 10 million, to the average of 31 million arc scans performed by the algorithm. This puts `PFP`+`KSM` close to the best `PR` code, which applies about 38 million arc scans and 2 million double pushes on average. Now, considering that the BFS based relabelings and lowest label searches are somewhat more cache efficient than the DFS based operations in `PFP`, we see that the operation counts are well comparable between the algorithms and are suited to gauge their performance.

The performance of the `PR` algorithm in terms of operation counts is also quite sensitive to the frequency of the global relabelings. A higher relabeling frequency means significantly more global relabel arc scans, but also a greatly reduced number of regular arc scans and double pushes. Using a least squares estimate on the experimental data, we asserted that in our implementation the computational cost of a double push is roughly equivalent to that of ten arc scans. Therefore, frequent global relabels are likely to pay off as long as they can substantially reduce the number of double pushes.

For the pseudoflow based codes, the operation count comparison is somewhat more difficult due to the nature of underlying complex operations. Nonetheless, the arc scan operation can still be used for comparisons. The low performance of the `LO_FREE` code compared to `PFP` and `PR` can be explained by the fact

Table 2: Average operation counts in millions over the entire test set. Detailed results are given for the various permutation types. AS denotes *arc scans* and DP denote *double pushes* in push relabel algorithms. For other algorithms, AS and the sum of other operations is given.

Algorithm	RF	No perm		Row perm		Col perm		Row + Col perm		Average	
		AS	DP	AS	DP	AS	DP	AS	DP	AS	DP
PR	1	28.95	1.32	59.25	2.38	149.94	4.58	81.88	2.02	79.82	2.57
SGM	1.5	26.93	1.13	60.28	2.34	123.77	3.98	70.30	2.06	70.22	2.38
	2	24.39	0.87	65.90	2.20	104.88	3.20	51.09	1.94	61.48	2.05
	4	24.40	0.62	63.69	1.69	93.55	2.19	47.25	1.44	57.13	1.49
PR-Fair	1	25.81	1.15	62.49	2.55	84.56	2.93	67.14	2.25	59.93	2.22
SGM	1.5	23.01	0.87	59.88	2.52	71.82	2.77	56.52	2.24	52.72	2.10
	2	24.57	0.83	60.65	2.27	63.64	2.29	47.56	1.97	49.07	1.84
	4	23.71	0.55	62.53	1.69	64.32	1.70	48.81	1.44	49.77	1.34
PRFair	1	23.68	1.00	47.97	2.55	48.38	2.70	33.40	2.09	38.34	2.08
-Spread	1.5	25.44	0.84	49.27	2.45	52.60	2.72	37.68	2.08	41.22	2.02
SGM	2	23.59	0.77	53.80	2.19	54.13	2.33	38.25	1.90	42.42	1.80
	4	23.37	0.52	63.00	1.65	62.95	1.69	46.95	1.42	49.03	1.32
Other algs.											
PFP		11.82	0.49	34.99	1.90	59.24	2.06	18.54	0.94	31.10	1.34
HI_WAVE		23.59	9.37	67.21	15.91	67.51	14.60	21.06	10.83	44.80	12.67
LO_LIFO		30.84	13.11	103.56	23.97	97.30	18.32	20.50	12.98	63.01	17.08
LO_FREE		130.49	24.40	27.12	8.38	157.14	31.99	16.92	6.47	83.04	17.82

that it performs about 82 million arc scans. However, HI_WAVE and LO_LIFO use significantly fewer arc scans than LO_FREE, yet their performance is inferior since the cost of the other operations is higher than that of the arc scans. Therefore, these counts cannot explain the difference in performance. The difference is due to the different trees generated by the corresponding split operations [15].

The operation counts are also well suited to indicate the difference between the PR algorithms. We observe that the fairness and the search spread mechanisms reduce the average number of operations. However, in the case of search spread, this benefit comes at the cost of slightly slower search in the adjacency lists, which is not captured by the operation counts. We also observe that using higher global relabeling frequencies universally reduces the number of double pushes. However, the effects on the total number of arc scans are varied.

6.5. Pseudoflow algorithms

We observe that the three pseudoflow algorithms perform comparatively poorly in this study. In contrast, their performance was found to be superior to the alternatives tested in [15], including the PR algorithm. This discrepancy cannot be explained with differences in initialization: the main algorithm's time for the pseudoflow algorithms is higher than the total time required for PR or PFP on the original matrices. Furthermore, we can assume that the pseudoflow codes do not suffer from insufficient algorithm engineering, since they were found [15] to be faster than known good push-relabel codes. A likely explanation can be found in the fact that Chandran and Hochbaum [15] use only the lowest-label variant of PR, not the FIFO variant for comparison, which was found to be substantially faster in our technical report [17].

To exclude the possibility of the results being a consequence of the different test sets, we performed an additional experiment using the *HiLo* and *rbg* random generators [21]. Similar to the largest bipartite graph instances in that study, our test cases have between 1.024 and 1.2 million vertices with average degrees of 5 or 10. Ten random instances were used for the test. Due to `KSM` initialization, `PFP` was extremely fast here, taking an average of only 0.95 seconds. `PR` with `KSM` initialization was equally fast. The `SGM` initialized `PR` codes took between 3.32 and 4.39 seconds, while `LO_FREE` took 9.86. `HI_WAVE` performed much better than `LO_FREE`, taking 5.64 seconds on average, while `LO_LIFO` took 23.45.

Based on the above results, we conclude that the difference in performance between `PR` and pseudoflow algorithms observed for real-world matrices is also evident in the mentioned random instances. In agreement with the previous results [15], `LO_FREE` was generally the fastest of the pseudoflow codes, followed by `HI_WAVE` and then `LO_LIFO`.

Finally, we note that in [15], a relatively old Sun UltraSPARC workstation with a 270 MHz CPU and 192 MB of RAM is used—this differs substantially from our test systems. However, the high operation counts of `LO_FREE` indicate that this difference cannot account for all the difference.

6.6. Fairness and spread mechanisms

For the original matrices, the fairness mechanisms in the `PR` algorithm showed little or no effect. However, for the column permuted matrices, the fair `PR` algorithms were significantly faster. In total, depending on the relabeling frequency, fairness improves the average running time by up to 30% (the median by 10%).

The search spreading technique yielded noticeable improvements in the average running time. It reduces the number of arc scans and double pushes. It also takes a small amount of extra running time. This manifests itself as a slight increase in median running time by about 3% w.r.t. `PR-Fair`, and a decrease in average running time by up to 30% due to much better running time on some hard instances. However, this technique requires more effort to implement; especially in combination with fairness (the next arc pointer costs additional memory space).

In conclusion, we recommend that `PR` should be implemented using both techniques. Their effects on running time noticeably outweigh the cost.

6.7. Relabeling frequency

For the average values, the relabeling frequency has little impact. Optimum RF values lie between 1 and 2. When using the search spread technique, the optimum value is 1. As observed in our technical report [17], when using techniques that improve the performance of the `PR` algorithm, relabeling frequency should generally be reduced.

Furthermore, it is interesting to note that for the original matrices, the highest relabeling frequency tested provided superior results, while on the permuted matrices, the lower frequencies generally work better.

We noticed that `PR` benefits from high relabeling frequencies in large matrices. However, this is due to the fact that the large matrices in the University

of Florida Sparse Matrix Collection are biased towards being more "difficult" (i.e., more time consuming for their size). Specifically, the recently introduced *DIMACS10* group contains many large and difficult matrices. A supplementary experiment on sparse uniformly distributed random matrices generated by Matlab's *sprand* command (i.e., bipartite Erdős-Rényi style graphs) of sizes between 2^{18} and 2^{23} columns and rows and an average degree of 3, 5, and 7 showed no correlation between matrix size and optimum global relabeling frequency. The experiment was performed on the Intel Xeon based secondary test system. Furthermore, as observed in the technical report [17], the optimum relabeling frequency for the random instances is higher than that for the real-world instances. Results can be found in the technical report [18, Table 7].

7. Concluding remarks

We have presented the adaptation of the push-relabel algorithm for bipartite matching and introduced simple yet effective techniques to improve its running time. Using the `FIFO` version, we have investigated its performance in comparison with the state-of-the-art augmenting path- and pseudoflow-based methods on real-world instances.

By experimenting thoroughly on a large number of problem instances arising in real-world applications, we drew several clear conclusions. We established that the augmenting path based algorithm `PFP` equipped with the `KSM` initialization heuristic is competitive with the `FIFO` variant of the `PR` algorithm using `SGM` initialization, and these two are preferable to other algorithms. Both are tied closely in running time and operation count, and their implementation requires comparable effort. However, using the additional techniques of spread and fairness, `PR` is slightly faster than `PFP`. Still, the difference between the augmenting and pushing approach is rather small. Furthermore, proper choice of initialization heuristics often has a greater impact than different algorithmic techniques. On the other hand, our results clearly show that on the real-world test instances used, the pseudoflow codes are not competitive with either `PR` or `PFP`.

References

[1] J. E. Hopcroft, R. M. Karp, An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, SIAM Journal on Computing 2 (4) (1973) 225–231.

[2] H. Alt, N. Blum, K. Mehlhorn, M. Paul, Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/\log n})$, Information Processing Letters 37 (4) (1991) 237–240.

[3] A. H. Timmer, J. A. G. Jess, Exact scheduling strategies based on bipartite graph matching, in: Proceedings of the 1995 European conference on Design and Test, EDTC '95, IEEE Computer Society, Washington, DC, USA, 1995, pp. 42–47.

[4] G. Lewandowski, Course scheduling: Metrics, models, and methods (1996).

[5] G. Lewandowski, P. Ojha, J. Rizzo, A. Walker, An average case approximation bound for course scheduling by greedy bipartite matching, in: Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling, 2002, pp. 144–147.

[6] W.-Y. Kim, A. Kak, 3-D object recognition using bipartite matching embedded in discrete relaxation, IEEE Transactions on Pattern Analysis and Machine Intelligence 13 (3) (1991) 224 –251. doi:10.1109/34.75511.

[7] P. E. John, H. Sachs, M. Zheng, Kekulé patterns and Clar patterns in bipartite plane graphs, Journal of Chemical Information and Computer Sciences 35 (6) (1995) 1019–1021.

[8] R. Burkard, M. Dell'Amico, S. Martello, Assignment Problems, SIAM, Philadelphia, PA, USA, 2009.

[9] T. A. Davis, E. Palamadai Natarajan, Algorithm 907: Klu, a direct sparse solver for circuit simulation problems, ACM Trans. Math. Softw. 37 (3) (2010) 36:1–36:17.

[10] T. A. Davis, Direct Methods for Sparse Linear Systems, SIAM, Philadelphia, PA, USA, 2006.

[11] I. S. Duff, A. M. Erisman, J. K. Reid, Direct Methods for Sparse Matrices, Oxford University Press, London, 1986.

[12] I. S. Duff, K. Kaya, B. Uçar, Design, implementation, and analysis of maximum transversal algorithms, ACM Transactions on Mathematical Software 38 (2011) 13:1–13:31.

[13] A. V. Goldberg, R. E. Tarjan, A new approach to the maximum flow problem, Journal of the Association for Computing Machinery 35 (1988) 921–940.

[14] D. S. Hochbaum, The pseudoflow algorithm and the pseudoflow-based simplex for the maximum flow problem, in: Proceedings of the 6th International IPCO Conference on Integer Programming and Combinatorial Optimization, Springer-Verlag, London, UK, 1998, pp. 325–337.

[15] B. G. Chandran, D. S. Hochbaum, Practical and theoretical improvements for bipartite matching using the pseudoflow algorithm, CoRR abs/1105.1569.

[16] T. A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 1:1–1:25.

[17] K. Kaya, J. Langguth, F. Manne, B. Uçar, Experiments on push-relabel-based maximum cardinality matching algorithms for bipartite graphs, Tech. Rep. TR/PA/11/33, CERFACS, France (2011).

[18] K. Kaya, J. Langguth, F. Manne, B. Uçar, Investigations on push-relabel based algorithms for the maximum transversal problem, Rapport de recherche RR-8093, INRIA (2012).

[19] L. Lovasz, M. D. Plummer, Matching Theory, North-Holland mathematics studies, Elsevier Science Publishers, Amsterdam, Netherlands, 1986.

[20] C. Berge, Two theorems in graph theory, Proceedings of the National Academy of Sciences of the USA 43 (1957) 842–844.

[21] B. V. Cherkassky, A. V. Goldberg, P. Martin, J. C. Setubal, J. Stolfi, Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms, Journal of Experimental Algorithmics 3 (1998) 8.

[22] A. Pothen, C.-J. Fan, Computing the block triangular form of a sparse matrix, ACM Transactions of Mathematical Software 16 (1990) 303–324.

[23] R. M. Karp, M. Sipser, Maximum matching in sparse random graphs, in: 22nd Annual IEEE Symposium on Foundations of Computer Science (FOCS 1981), IEEE Computer Society, Los Alamitos, CA, USA, 1981, pp. 364–375.

[24] J. Langguth, F. Manne, P. Sanders, Heuristic initialization for bipartite matching problems, ACM Journal of Experimental Algorithmics 15 (February, 2010) 1.1–1.22.

[25] J. Magun, Greedy matching algorithms, an experimental study, Journal of Experimental Algorithmics 3 (1998) 6.

[26] J. C. Setubal, Sequential and parallel experimental results with bipartite matching algorithms, Tech. Rep. IC-96-09, Univ. of Campinas, Brazil (September 1996).

[27] J. Aronson, A. Frieze, B. G. Pittel, Maximum matchings in sparse random graphs: Karp-Sipser revisited, Random Structures and Algorithms 12 (1998) 111–177.

[28] R. J. Kennedy, Jr., Solving unweighted and weighted bipartite matching problems in theory and practice, Ph.D. thesis, Stanford University, Stanford, CA, USA, uMI Order No. GAX96-02908 (1995).

[29] K. Mehlhorn, S. Näher, LEDA: A Platform for Combinatorial and Geometric Computing, Cambridge University Press, Cambridge, UK, 1999.

[30] I. S. Duff, On algorithms for obtaining a maximum transversal, ACM Transactions on Mathematical Software 7 (1981) 315–330.

[31] B. G. Chandran, D. S. Hochbaum, A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem, Operations Research 57 (2) (2009) 358–376.